Bypass and Insertion Algorithms for Exclusive Last-level Caches

Jayesh Gaur
Intel Architecture Group
Intel Corporation
Bangalore 560103
INDIA
jayesh.gaur@intel.com

Mainak Chaudhuri*
Department of CSE
Indian Institute of Technology
Kanpur 208016
INDIA
mainakc@cse.iitk.ac.in

Sreenivas Subramoney
Intel Architecture Group
Intel Corporation
Bangalore 560103
INDIA
sreenivas.subramoney@intel.com

ABSTRACT

Inclusive last-level caches (LLCs) waste precious silicon estate due to cross-level replication of cache blocks. As the industry moves toward cache hierarchies with larger inner levels, this wasted cache space leads to bigger performance losses compared to exclusive LLCs. However, exclusive LLCs make the design of replacement policies more challenging. While in an inclusive LLC a block can gather a filtered access history, this is not possible in an exclusive design because the block is de-allocated from the LLC on a hit. As a result, the popular least-recently-used replacement policy and its approximations are rendered ineffective and proper choice of insertion ages of cache blocks becomes even more important in exclusive designs. On the other hand, it is not necessary to fill every block into an exclusive LLC. This is known as selective cache bypassing and is not possible to implement in an inclusive LLC because that would violate inclusion. This paper explores insertion and bypass algorithms for exclusive LLCs. Our detailed execution-driven simulation results show that a combination of our best insertion and bypass policies delivers an improvement of up to 61.2% and on average (geometric mean) 3.4% in terms of instructions retired per cycle (IPC) for 97 single-threaded dynamic instruction traces spanning selected SPEC 2006 and server applications, running on a 2 MB 16-way exclusive LLC compared to a baseline exclusive design in the presence of well-tuned multi-stream hardware prefetchers. The corresponding improvements in throughput for 35 4-way multi-programmed workloads running with an 8 MB 16-way shared exclusive LLC are 20.6% (maximum) and 2.5% (geometric mean).

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Design Styles

General Terms

Algorithms, design, measurement, performance

* The work was done while the author was a visiting researcher at Intel India.

ISCA'11, June 4–8, 2011, San Jose, California, USA.

Keywords

Exclusive last-level cache, bypass policy, insertion policy

1. INTRODUCTION

Inclusive last-level caches (LLCs) simplify cache coherence protocol; an LLC tag lookup is enough to decide if a cache block is not present in the inner levels of the cache hierarchy. In an exclusive LLC, however, a block is allocated only on an eviction from the inner level cache and de-allocated on a hit when the block is recalled by the inner level cache.[1] As a result, a separate coherence directory array (decoupled from the LLC tag array) is needed to maintain coherence efficiently. While coherence simplification and silent clean evictions from the inner level are seen as major advantages of an inclusive LLC, such a design, by definition, wastes silicon estate due to replication of cached data in multiple levels of the hierarchy. As the industry moves toward a three-level or a four-level cache hierarchy with reasonably large inner levels, such cross-level replication begins to hurt performance in an inclusive design when compared to an exclusive one. This observation has already motivated commercial processor designers to adopt fully or partially exclusive LLCs [1].

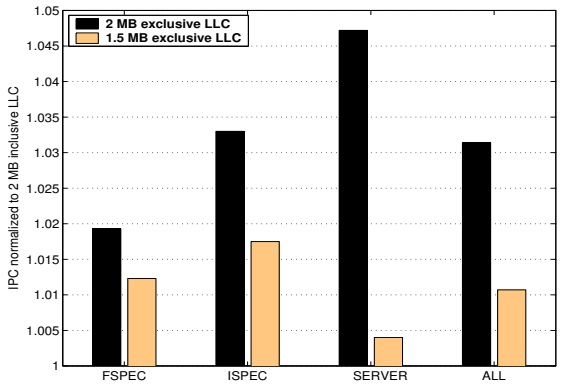


Figure 1: A comparison of IPC between exclusive and inclusive LLCs with a 512 KB L2 cache in each design.

The performance gains in an exclusive design over an identical inclusive design usually come from two factors. One of these is the overall capacity advantage enjoyed by the exclusive design. The second performance factor is related to premature evictions in the inner levels of the hierarchy

[1] To enable fast cross-thread access to shared data, shared blocks may not be de-allocated on hits. We leave the exploration of replacement policies for shared blocks in such a non-inclusive/non-exclusive design to future work.

caused by LLC replacements in an inclusive design. In the absence of access hints from the L1 and L2 caches, the last level (L3 in this study) of an inclusive design can end up making wrong replacement decisions [14]. The risk of premature evictions from the L1 and L2 caches triggered by LLC replacements is non-existent in an exclusive design.

In Figure 1, we show the performance of an exclusive LLC relative to an inclusive LLC for 97 single-threaded dynamic instruction traces representing different regions of floating-point SPEC 2006 (FSPEC), integer SPEC 2006 (ISPEC), and server (SERVER) applications with a well-tuned multi-stream hardware prefetcher enabled. For the left bar, the simulated three-level cache hierarchy in both inclusive and exclusive cases is identical in capacity and associativity at every level. More specifically, the left bar presents simulation results for an architecture with a 512 KB 8-way L2 cache and a 2 MB 16-way LLC. The bar on the right shows the performance of an exclusive LLC relative to an inclusive design, where the exclusive LLC is sized (1.5 MB 12-way) such that the effective capacity advantage of the exclusive design is nullified. In both cases, the inclusive LLC simulates a not-recently-used (NRU) replacement policy (one bit age per block) and the exclusive LLC simulates a one-bit not-recently-filled (NRF) replacement policy. The NRU policy victimizes the not recently used block (age zero) from the way with the smallest id. The NRF policy updates the age bit array only on a fill and is otherwise similar to NRU. Both the policies reset the age bits of all the blocks (except the one most recently accessed/filled) in a set to zero only if all blocks in that set have age of one. For each application category, the bar on the right in Figure 1 brings out the performance difference stemming from the premature evictions from the inner levels of the cache hierarchy in the inclusive design. The bar on the left further adds the capacity advantage that an exclusive design enjoys. On average, for 97 traces, the exclusive design enjoys a 3.1% higher IPC than the inclusive design.

While premature LLC replacements can cause performance degradation in inclusive designs, a block resident in the LLC of an inclusive design can still observe a filtered version of access recency during its life time. The situation can further improve with access hints from the inner levels or other proactive mechanisms [14]. This is not possible in an exclusive design. A block resides in the LLC of an exclusive design from the time it is evicted from the L2 cache to the time it is either recalled by the L2 cache or evicted by the LLC, whichever is earlier. Due to the absence of any access information in an exclusive LLC, the popular least-recently-used (LRU) replacement policy and all its well-researched derivatives lose significance. As a result, the design of replacement policies in an exclusive LLC requires a fresh look. A replacement policy has three distinct algorithmic components, namely, insertion age algorithm, age update algorithm, and victim selection algorithm. In this paper, we explore the insertion age algorithms.

Selective bypass is an important optimization that can be exercised in an exclusive design, since every block evicted from the L2 cache is not required to be filled into the LLC. In an inclusive design, all blocks fetched from memory must be filled in the LLC to maintain inclusion. We explore LLC bypass algorithms in this paper for an exclusive design that can identify clean as well as modified blocks that need not be filled in the LLC. Good LLC bypass policies can improve performance in two ways, namely, by reducing the bandwidth demand of the on-die interconnect as well as the LLC controller and by allocating the LLC capacity only to the blocks with relatively short reuse distances.

1.1 Motivation

Figure 2 motivates the two design problems, namely, bypass and insertion in exclusive LLCs, that we explore in this paper. In this figure, we consider a baseline exclusive LLC with NRF replacement policy and no bypass (i.e., all L2 cache evictions are allocated in the LLC). We keep the hardware prefetchers turned off to gain a better understanding of the demand request behavior (we will present results with prefetchers turned on in Section 5). The experiments are conducted on a single-core system with a 2 MB 16-way exclusive LLC. The L2 cache is 512 KB 8-way set associative and instruction and data L1 caches are 16 KB 4-way and 32 KB 8-way set associative, respectively. The L1 and L2 caches execute a pseudo-LRU replacement policy.

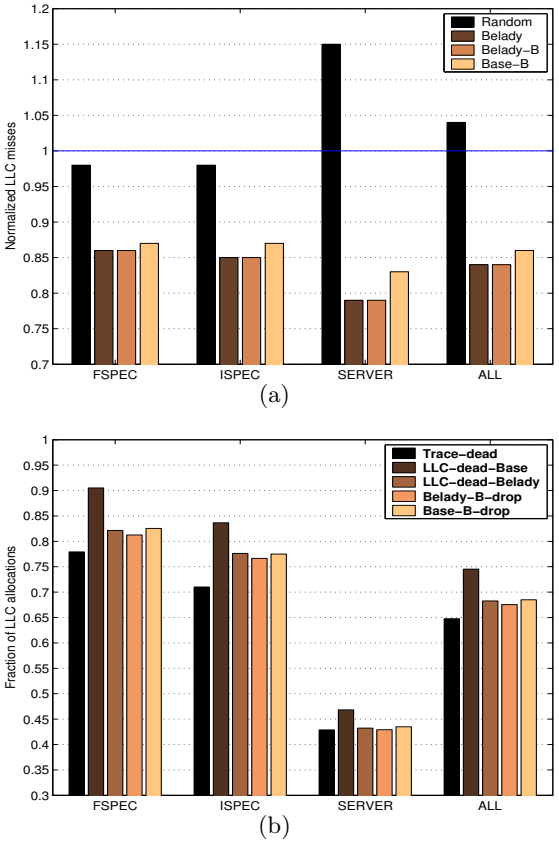


Figure 2: (a) Normalized LLC misses for random replacement and a number of oracle-assisted replacement policies. (b) Dead allocation and bypass analysis for a number of oracle-assisted bypass policies.

Figure 2(a) presents the number of LLC misses normalized to the baseline NRF policy for a number of schemes, namely, the popular random replacement with no bypass often used for victim caches (Random), Belady's optimal longest-forward-distance replacement [3, 26] with no bypass (Belady), Belady's optimal replacement extended with bypass which drops an incoming block if its next forward use distance is larger than all the blocks in the target LLC set (Belady-B), and the baseline policy extended with optimal bypass which drops an incoming block if its next forward use distance is larger than the current victim in the target LLC set (Base-B). In this paper, we never bypass instruction blocks. All these experiments are done on an offline cache simulator that has access to the entire LLC allocation/lookup trace. The cache state at the end of warmup is loaded from a checkpoint. The tie between blocks with "unknown" forward distance (next potential use is beyond the trace length) is broken arbitrarily.

These results show that random replacement leads to 4% more LLC misses compared to the baseline policy. On the

other hand, Belady, Belady-B, and Base-B show significant potential for improvement. On average, these three schemes save 16%, 16%, and 14% LLC misses.[2] It is interesting to note that Belady and Belady-B are equally effective meaning that augmenting an already good replacement policy with a bypass decision does not bring any extra advantage in terms of LLC hits. In this case, even though a bypass policy cannot improve the volume of LLC hits, it can save a significant amount of on-die interconnect bandwidth if one could implement a bypass scheme at the L2 cache boundary. It is, however, encouraging to note that a forward-looking bypass scheme working with the NRF policy (Base-B) can save 14% of LLC misses. As expected, the potential for hit rate improvement of a good bypass scheme increases as the replacement policy gets inferior. We, however, make a note of the fact that Base-B comes surprisingly close to Belady and Belady-B, the gap being maximized in the server workloads. We will explain this with the help of Figure 2(b).

Figure 2(b) shows additional data pertaining to the bypass potential in an exclusive design. For each trace category, Figure 2(b) shows five different statistics: 1) the number of blocks allocated in the LLC but not used again in the rest of the trace as a fraction of all allocations (Trace-dead), 2) the fraction of blocks allocated in the LLC that are not recalled by the L2 cache before getting evicted from the baseline LLC (LLC-dead-Base), 3) the fraction of blocks allocated in the LLC that are not recalled by the L2 cache before getting evicted from the LLC while executing optimal replacement with no bypass (LLC-dead-Belady), 4) the fraction of all L2 cache evictions bypassed by Belady-B (Belady-B-drop), and 5) the fraction of all L2 cache evictions bypassed by Base-B (Base-B-drop). Among these, the first bar shows the fraction of useless allocations that would have happened in an LLC of infinite capacity. Although this fraction is a direct function of the trace length (longer traces are likely to have smaller values of this fraction), this data emphasizes the facts that the L1 and L2 caches are doing a wonderful job in absorbing all short-term reuses and that most of these reuse clusters are located very far apart. The next bar shows the fraction of useless LLC allocations in the baseline LLC. This is about 75%, on average. This fraction is a realistic representation of the bypass potential. A bypass algorithm should try to make room in the LLC so that a subset of the blocks contributing to the difference of the first two bars can be retained. The third bar shows that even an optimal replacement policy does about 68% of useless allocations in the LLC. This result underscores the importance of a good bypass policy even if the replacement policy is optimal. The fourth bar further confirms this result by showing an equivalent volume of bypasses that an optimal replacement policy with optimal bypass would observe. This result brings to fore the large potential of on-die interconnect bandwidth saving that an LLC bypass policy running at the L2 cache boundary can achieve. Finally, the rightmost bar shows that a forward-looking bypass policy executing with the baseline replacement policy can be as effective as optimal bypass and replacement running together in terms of the fraction of bypassed blocks. We have already noted that the performance gap between Base-B and Belady-B is maximum for the server workloads. This is expected since the bypass fraction is minimum in these workloads leading to the invocation of suboptimal NRF replacement of Base-B for a significant fraction of fills.

Given the large potential in terms of replacement and bypass algorithms, this paper systematically deduces and implements a few such algorithms (Sections 2 and 3). In this study, we explore only the insertion component of the replacement algorithms. Although our bypass algorithms can be seamlessly integrated with the L2 cache, we implement them in the LLC controller and as a result, explore only the capacity benefit of these algorithms. Our detailed execution-driven simulation results (Sections 4 and 5) show that the combination of our best insertion and bypass algorithms improves the IPC of 97 single-threaded traces by up to 61.2% and on average 3.4% on a 2 MB 16-way exclusive LLC compared to the baseline exclusive design with aggressive multi-stream hardware prefetchers enabled. The corresponding maximum and average improvements in throughput for 35 4-way multi-programmed workloads are 20.6% and 2.5%.

1.2 Related Work

In this section, we briefly review the studies that are most relevant to our work. The capacity advantage of exclusive LLCs compared to inclusive LLCs has been studied in [29], while several techniques to get rid of premature evictions from inner level caches in an inclusive hierarchy have been proposed in [14]. The importance of treating shared blocks specially in an exclusive LLC has been highlighted in [24]. Multi-level exclusive caching in the context of distributed disk caches has been studied in [7].

Exclusive caches are functionally equivalent to large victim caches [18]. Selective victim caching (analogous to an exclusive LLC with bypass enabled) has been explored in the context of small victim caches that work well with L1 caches [5, 11]. A design of a large victim cache with selective insertion based on frequency of misses has been presented in [2] and is shown to work well with inclusive LLCs. A recent work exploits the dead blocks in an inclusive LLC to configure an "embedded" victim cache [21].

Dead block prediction schemes [11, 19, 21, 22, 23, 25] have close connection with our bypass proposal. Most of the existing dead block prediction schemes select a dead block in a cache set either for replacement or as a target of a prefetch after the block has spent some time in the cache. These proposals usually correlate the instruction address and/or data address with the death of a cache block. A recent proposal shows how to design an address-oblivious dead block predictor that exploits the reuse probabilities to improve the replacement decisions in an inclusive LLC [4]. On the other hand, our bypass algorithms identify a block that would be dead-on-fill in the LLC at the time of fill. While instruction or data address can improve the quality of bypass, our bypass algorithms do not rely on any such information.

A few recent proposals have explored bypass algorithms for LLCs.[3] One proposal [6] remembers the tag of a bypassed block (if the incoming block is bypassed) or the victimized block (if the incoming block is not bypassed) to observe the next use to the bypassed/allocated block and the saved/victimized block, and accordingly learns whether bypassing is a good decision. However, this proposal randomly selects incoming blocks for bypassing based on a bypass probability, which is adjusted dynamically based on the effectiveness of bypassing. Another proposal [19] shows how to use a program counter (PC)-based skewed dead block predictor that learns the caching behavior of a few sample sets in the LLC to identify blocks that are dead-on-fill and bypass such blocks. Access counter-based LLC bypass algorithms that take help of a prediction table indexed by a hash function of the PC are explored in [22]. Our bypass algorithms do not require separate prediction tables or any PC-related information, but exploit the reuse frequency that a block sees while in the L2 cache and the number of times it has traveled between the L2 cache and the exclusive LLC. It is important to note that bypass algorithms have also been studied in the context of small data caches.

[2] The optimal replacement and optimal bypass gains may change for a different interleaving of LLC requests.

[3] Often these proposals do not clearly mention the nature of the cache hierarchy being considered. We assume that these are done in the context of exclusive or partially inclusive caches.

These proposals require a profile pass that gathers the locality information [17], or build a PC and data address-based locality predictor [8], or carry out an instruction-based characterization of the potential of data cache bypassing [28], or employ a classification of data cache misses into capacity or conflict to drive the bypass decision [5]. Our proposal does not require any profile run or any PC/address/instruction information or miss classification.

There have been several recent studies on insertion age selection for LLCs. Some of these studies require PC information of the source instruction of a to-be-filled block [9, 13]. Also, several of these studies usually identify the insertion age with LRU, MRU, or other access recency positions in a set [9, 13, 16, 20, 27]. As a result, these proposals get tied to the notion of an access recency order, which is nonexistent in an exclusive cache (in the absence of any extra information, the only meaningful order among the blocks in an exclusive cache set is the fill order). A recent study assigns insertion age based on re-reference interval prediction and updates the predicted age on a hit in an inclusive LLC [15]. Although such an option of age update is nonexistent in an exclusive LLC, we will show how to design somewhat analogous policies for exclusive LLCs. A decision tree-based technique for selecting the insertion age relative to the access recency order has been explored in [20]. Our insertion policy proposals are necessarily independent of access recency order.

2. CHARACTERIZATION OF DEAD AND LIVE LLC BLOCKS

Use recency and use frequency are the two properties that are traditionally employed to determine the death and liveness of a cache block. Exclusive LLCs make the effective use of these two properties challenging because a block is de-allocated from the LLC on its first recall from the inner-level caches. As a result, the only meaningful order among the blocks in an LLC set is the fill order. Unfortunately, the fill order among the blocks in an LLC set has little correlation with their use recency order because blocks evicted from multiple L2 cache sets may get allocated in the same LLC set. Even though the L2 cache exercises a use recency based replacement algorithm (e.g., pseudo-LRU in our case), the fill order in an LLC set is an arbitrary interleaving of the use recency orders of multiple L2 cache sets from different cores or the same core. Reconstructing the correct use recency order in an LLC set is costly, since it requires a global use recency order in the L2 caches across multiple cores. In this paper, we design our bypass and insertion schemes based on estimates of average recall distance of LLC blocks and their use count in the L2 cache. The average recall distance of an LLC block B is defined as the mean number of LLC allocations between the allocation of B in the LLC and the recall of B from the L2 cache.

2.1 Estimate of Recall Distance

In a three-level cache hierarchy with an exclusive LLC, a block is filled into the L2 cache when it is first brought from the DRAM. On an L2 cache eviction, it makes its first trip to the LLC. If it is recalled from the LLC before it is evicted, it will eventually make its second trip to the LLC when it is victimized from the L2 cache again. These trips continue until the block is evicted from the LLC. A block with a high trip count is expected to have a low average recall distance. The trip count of a block in an exclusive LLC translates to the use count of the block in an inclusive LLC.

To understand the trip count behavior in the presence of optimal LLC replacement decisions, Figure 3(a) shows the distribution of LLC allocations among four trip count (TC) bins (the first trip from the L2 cache to the LLC is denoted by TC=0). While every block brought into the cache hierarchy gets allocated in the LLC for the first time with TC=0, only a fraction of that will survive to experience a TC=1 allocation. The difference between TC=0 and TC=1 allocation fractions brings out the percentage of useless allocations that happen at TC=0. Overall, this is about 55% of all LLC allocations. We note that this forms a major portion of the overall useless allocation fraction of 68% (see LLC-dead-Belady in Figure 2(b)). We will refer to the blocks allocated with TC=0 as the TC_0 blocks and the rest as $TC_{\geq 1}$ blocks.

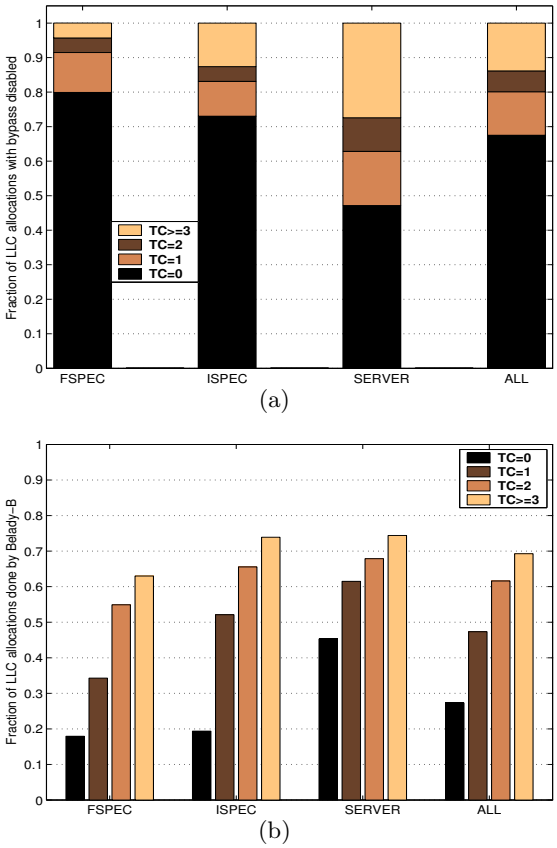


Figure 3: (a) Distribution of LLC allocations with trip count in the presence of optimal replacement with no bypass (Belady) in the LLC. (b) Fraction of LLC allocations in each trip count bin in the presence of optimal replacement with optimal bypass (Belady-B) in the LLC.

Figure 3(b) further shows the fraction of allocations that take place in each TC bin when an optimal bypass is enabled on top of optimal replacement (Belady-B) in the LLC. This fraction for a particular bin is computed as the number of LLC allocations made from that bin over the number of incoming blocks belonging to that bin. Note that the incoming blocks belonging to the TC=k bin are necessarily a subset of the blocks allocated from the TC=$k-1$ bin for $k \geq 1$ (the remaining subset gets evicted from the LLC before they can make the next trip). These data show that overall, only 27% of the TC_0 blocks are allocated in the LLC and the remaining 73% of the TC_0 blocks are bypassed. The allocation percentage progressively increases as a block moves to higher TC bins. These data clearly bring out the fact that the likelihood of an LLC block being live increases with its TC value (up to a limit). We derive three major conclusions from these data. First, the TC=0 bin is the ideal bypass target and the $TC_{\geq 1}$ blocks should be mostly allocated in the LLC. However, it is important to separate the dead TC_0 blocks from the live TC_0 blocks (these live TC_0 blocks will eventually become $TC_{\geq 1}$ blocks). We explore L2 use count in the next section as a possible feature to carry out this

classification. Second, since $TC_{\geq 1}$ blocks are mostly live, they can be assigned a high insertion age, if they are not bypassed. However, we need more properties of the non-bypassed TC_0 blocks to appropriately grade their insertion ages. For this purpose, we explore L2 cache use count in the next section. Finally, two TC bins, namely, TC_0 and $TC_{\geq 1}$ are enough to derive most of the benefits. Therefore, we need one bit per L2 cache block and no storage for LLC blocks for maintaining the trip count (a block recalled from the LLC is always classified as $TC_{\geq 1}$ in the L2 cache). More TC bits would help the server traces, but the overall utilization of these bits would be low.

2.2 Use Count and Synergy with Trip Count

In the last section, we have seen that trip count can serve as a reasonably good starting point for identifying a large fraction of bypass candidates (a major portion of TC_0 blocks) and live blocks in exclusive LLCs. In the following, we explore the possibility of exploiting the use count of a block during its residency in the L2 cache to further tune the classification of dead and live blocks. Every time a block is filled into the L2 cache (from DRAM or LLC) by a demand request, its use count is set to one. A block filled into the L2 cache by a prefetch request sets its use count to zero. Only a demand hit in the L2 cache can increment the use count. While the trip count is loosely related to the average distance between clusters of near-term reuses to a block, the L2 cache use count captures the size of the last such cluster seen by a block. In Figure 4(a), we start by exploring the distribution of LLC allocations among three L2 cache use count (L2UC) bins. The LLC executes Belady's optimal replacement policy with no bypass. Note that in the absence of prefetching, a block evicted from the L2 cache cannot have a zero L2UC. Even in the presence of prefetching, such blocks may exist only due to premature or incorrect prefetches. In Section 3.4, we will discuss how to handle such blocks when they come for LLC allocation. The data in Figure 4(a) show that about 58% of blocks allocated in the LLC observe only a single use in the L2 cache. The next L2UC bin contributes about 30% of LLC allocations. The remaining allocations come with at least three L2UC. While these data do not offer any insight into classification of dead and live blocks, they do confirm that two bits for maintaining L2UC per L2 cache block is enough for all practical purposes. Recognizing the fact that L2UC is only a filtered (or sampled) access count, we also looked at the cumulative use count (CUC) of a block in the L1 cache counted from the time the block is brought into the L1 cache till the time it is evicted from the L2 cache (this corresponds to L1 cache use count per L2-LLC trip). We found that we need four bits per cache block to properly maintain CUC. In the following, we will explore if the added accuracy in CUC (compared to L2UC) can help improve the classification of dead and live blocks.

Design of good bypass and insertion age assignment algorithms requires understanding of the distribution of dead and live blocks in an optimal setting. One way to explore this is to observe the distribution of victims when the LLC executes Belady's optimal replacement because optimal victim selection is synonymous to optimal death prediction. To understand the distribution of good LLC victims, we execute Belady's optimal replacement in the LLC with bypass disabled while maintaining three L2UC bins (L2UC=0 is excluded), fifteen CUC bins (CUC=0 is excluded), and the cross-product of the TC bins with the L2UC and CUC bins i.e., six TC×L2UC bins and thirty TC×CUC bins. This creates four bin classes, namely, L2UC, CUC, TC×L2UC, and TC×CUC. We would like to know which of these bin classes could serve as a good feature for identifying dead and live blocks. One way to resolve this question is to identify, within each bin class, the bin with the maximum number of victims. Clearly, such a bin would capture most of the optimal victims. However, we want a low volume of live blocks in that bin so that the likelihood of victimizing live blocks is minimized. The goal is to identify the bin class that has a bin which maximizes the victim coverage and minimizes the live coverage. To achieve this, we do the following.

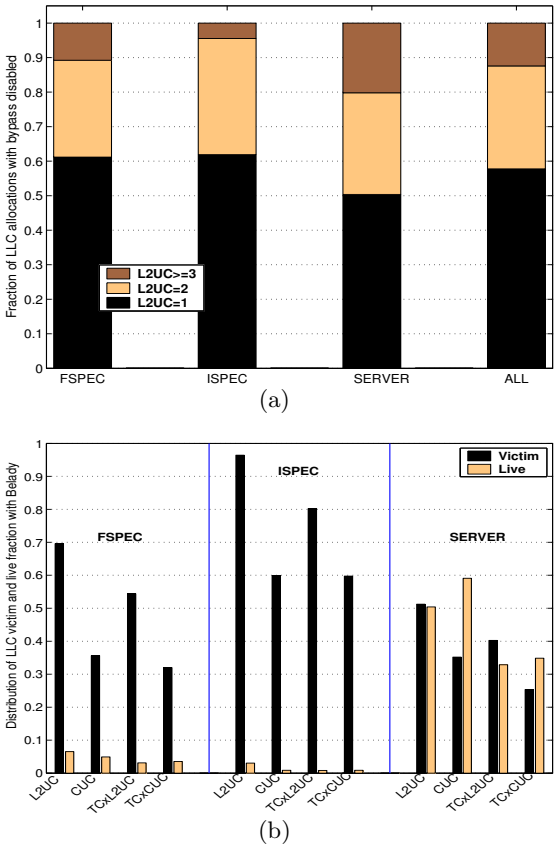


Figure 4: (a) Distribution of LLC allocations with L2 cache use count. (b) Median of victim and live block fractions in the most prominent victim bin for four bin classes. The victim fraction is the victim count of the most prominent victim bin out of all LLC victims across all bins, while the live fraction is computed over the LLC allocations done from the most prominent victim bin only. The data for (a) and (b) are collected in the presence of optimal replacement with no bypass (Belady) in the LLC.

When a block is allocated into the LLC, its membership bin in each of these four bin classes is decided based on its TC, L2UC, and CUC values. For example, a block with TC=0, L2UC=2, and CUC=10 will fall into L2UC=2 bin, CUC=10 bin, TC×L2UC=(0, 2) bin, and TC×CUC=(0, 10) bin. When a block is victimized from the LLC by the optimal replacement policy, the victim counts of the block's four membership bins are each incremented by one. For each trace, we identify the bin covering the maximum fraction of victims ($V^C_{\max}$) in each of the four bin classes C with $C \in$ {L2UC, CUC, TC×L2UC, TC×CUC}. For different traces, these four identified bins (one in each bin class) may turn out to be different. For each of these four identified bins (one in each bin class C), we also record the live fraction (L^C), computed as the number of LLC hits experienced by blocks belonging to a bin over the number of LLC allocations done from that bin.

Figure 4(b) presents the median of $V^C_{\max}$ and L^C for each of the three trace categories for each bin class C. The victim fraction is maximized in L2UC with TC×L2UC follow-

ing next, while the live fraction is minimized in TC×L2UC across the board, especially for the server traces. In the server traces, the gap in the live fraction between L2UC and TC×L2UC is much bigger than the gap in the victim fraction. Assuming that minimizing the live fraction is an equally important objective as maximizing the victim coverage, we decide to use the TC×L2UC bins for inferring the bypass candidates and insertion ages. We will refer to these bins as the TC-UC bins and L2UC as UC. It is encouraging to note that a high median victim fraction in TC×L2UC across application categories essentially indicates the existence of at least one TC-UC bin for each trace such that the dead blocks have a strong affinity toward it (with a membership likelihood of 0.55 for FSPEC, 0.80 for ISPEC, and 0.40 for SERVER). Also, the likelihood of misclassifying a live block belonging to such a bin as a dead block is negligibly small in FSPEC (0.03) and ISPEC (less than 0.01), while for SERVER it is about one-third. Our algorithms attempt to learn this bin and any other prominent dead bins dynamically. Even though the statistics presented in Figure 4(b) summarize the aggregate observed behavior for each trace, it is important to note that the prominent dead bins can change over time within the same application trace.

3. BYPASS AND INSERTION POLICIES

This section discusses the design and implementation of the bypass and insertion algorithms for exclusive LLCs. First we discuss the dynamic learning framework that all our algorithms use and then present the algorithms.

3.1 General Framework

The bypass and insertion decisions should be based on the population of dead and live blocks in the TC-UC bins. Note that a block allocated in the exclusive LLC is classified as dead if it gets evicted before getting recalled by the L2 cache (these are essentially LLC victims); otherwise the block is classified as live. Ideally, we would like to learn the dead and live populations in each TC-UC bin. Depending on the membership bin of an incoming block and the dead and live populations of that bin, we would like to take a decision about whether to bypass this block or what initial age to assign if it is not bypassed. To carry out this learning, we dedicate sixteen sample sets per 1024 sets of LLC that observe the dead and live populations of each TC-UC bin. These sets will be referred to as the observers. The observers allocate all blocks and implement a static insertion age assignment scheme based on the single-bit TC value of an incoming block. We will introduce this age assignment scheme in Section 3.3.

For each TC-UC bin per LLC bank, the observers maintain two values, namely, the difference of dead and live allocations to the observers ($D-L$) and the live allocations to the observers (L). Our algorithms need eight $D-L$ and eight L counters per LLC bank corresponding to the eight TC-UC bins. When a block arrives at the LLC for allocation to one of the observers, the block's TC-UC bin b is decided based on the block's TC, UC values (carried by the eviction message from the L2 cache). The observer increments the $D-L$ counter of bin b by one when the block is allocated. On a hit to a block B in an observer set, the observer decrements the $D-L$ counter of the bin the block B belongs to by two and increments that bin's L counter by one. The observers maintain three bits per cache block to remember the bin an allocated block belongs to. The non-observer sets, however, do not need to store any such information. A non-observer set, when allocating a block, first determines the block's membership bin based on the block's TC, UC values and then queries the $D-L$ and L counters of that bin. The returned $D-L$ and L values are input to the bypass and insertion algorithms.

When updating the $D-L$ and L counters in an LLC bank, the observers also maintain the $\max(D-L)$, $\min(D-L)$, $\max(L)$, and $\min(L)$ across the TC-UC bins, excluding the UC=0 bins, within that LLC bank. In addition to these, the aggregate $D-L$ over all TC-UC bins, excluding the UC=0 bins, is maintained per LLC bank. We will refer to this as $\sum_{UC\neq 0}(D-L)$. One of our insertion algorithms requires that the observers maintain the aggregate L over all TC=0 bins with positive UC. We will refer to this aggregate as $\sum_{TC=0,UC\neq 0}(L)$. The updates of the maximum, minimum, and the aggregate values take place mostly off the critical path of LLC activities. Every N LLC allocations per bank all the $D-L$ and L counters (including the max, min, and aggregate values) in that LLC bank are halved so that a temporally-aware exponential average is maintained. N is equal to the number of observer sets per LLC bank multiplied by the LLC associativity. Even with a storage overhead of two bytes per counter, the overall counter overhead is small. Our simulations use eight-bit $D-L$ and L counters for the single-threaded configuration and nine-bit counters for the multi-programmed configuration. The max, min, and aggregate registers are sized accordingly. Also, every L2 cache block stores three additional bits to maintain the TC and UC values of the block.

3.2 Bypass Algorithms

Good bypass algorithms would bypass incoming blocks that belong to bins with high $D-L$ populations, yet low enough L populations. More specifically, an incoming block belonging to TC-UC bin b with counter values $(D-L)_b$ and L_b qualifies as a bypass candidate if $(D-L)_b \geq \frac{1}{2}(\max(D-L)+\min(D-L))$ and $L_b \leq \frac{1}{2}(\max(L)+\min(L))$. However, we find that there are situations where the overall magnitude of $D-L$ is so high that even if the second condition fails, bypassing can be done without any performance degradation. Therefore, we override the outcomes of these comparisons if $(D-L)_b \geq \frac{3}{4}\sum_{UC\neq 0}(D-L)$. A more carefully chosen weight of magnitude lower than $\frac{3}{4}$ may improve the bypass performance further. We summarize our bypass algorithm in the following where `bypass` is a boolean-valued variable.

$$\begin{aligned} bypass \quad = \quad & ((D-L)_b \geq \frac{1}{2}(\max(D-L)+\min(D-L)) \\ & \text{AND } L_b \leq \frac{1}{2}(\max(L)+\min(L))) \\ & \text{OR} \quad ((D-L)_b \geq \frac{3}{4}\sum_{UC\neq 0}(D-L)) \end{aligned} \tag{1}$$

If an incoming block finds an invalid way in the target set and `bypass` is true according to the above formula, it is filled into the LLC with an insertion age of zero. In other words, invalid ways in the LLC are always utilized, but with a zero insertion age in the case a bypass is recommended. On the other hand, if `bypass` is true and there is no invalid way in the target set, the incoming block is bypassed. A bypassed block is treated exactly the same way as an LLC victim and it mimics the LLC eviction protocol.

To minimize the risk of performance loss, we employ dueling on set samples [27] and always duel our bypass algorithm with the no-bypass algorithm of the observers. For this purpose, in addition to the observer sets, we dedicate an equal number of LLC sets (sixteen per 1024 LLC sets) that always execute our bypass algorithm. We have observed that the static version of the bypass policy that does not employ any dueling degrades the performance of several applications.

3.3 Insertion Algorithms

We present three algorithms for insertion age assignment with progressively increasing complexity. These algorithms

are applied to those blocks for which `bypass` is false as computed by Formula (1). We assume a two-bit budget to maintain ages per LLC block. Our algorithms apply to data blocks only and instruction blocks are always filled with the highest age i.e., three. Our LLC replacement policy first looks for an invalid way in the target set. If there is no such way, it victimizes the block with the minimum age and also decrements all the ages in that set by this minimum before the new block is inserted to reflect the correct relative age order. A tie among the blocks with the minimum age is broken by selecting the block with the least physical way id.

Our first insertion algorithm is inspired by the distribution of liveness in TC_0 and $TC_{\geq 1}$ blocks as shown in Figure 3(b). This algorithm assigns all $TC_{\geq 1}$ blocks an insertion age of three and all TC_0 blocks an insertion age of one. This is the policy exercised by our observer sets, since it does not require any dynamic learning. We will refer to this policy as the TC-AGE policy. The TC-AGE policy is similar to SRRIP-style static algorithms originally proposed for inclusive LLCs [15]. In our age assignment setting where a lower age corresponds to a higher replacement priority, the SRRIP algorithm would assign an insertion age of one to a newly allocated block and promote it to the highest possible age on a hit in an inclusive LLC. In an exclusive LLC, the blocks that have already seen LLC hit(s) are the $TC_{\geq 1}$ blocks.

Our second insertion algorithm continues to assign the highest age, i.e., three to the $TC_{\geq 1}$ blocks, but it assigns more finely graded ages to the TC_0 blocks. To achieve this, it takes help of the dead and live populations learned by the observers. This algorithm recognizes the fact that the TC_0 blocks belonging to bins with low hit rates should not get a positive age. If a certain bin b satisfies $D_b > xL_b$ or equivalently, $(D-L)_b > (x-1)L_b$, that would translate to a hit rate bounded above by $\frac{1}{x+1}$ for blocks belonging to bin b (hit rate is $\frac{L_b}{D_b+L_b}$). We would like to assign an insertion age of zero to an incoming block if it belongs to a TC=0 bin with too low a hit rate. However, we find that there are situations where the hit rate of the target bin is low, but the bin still has a fairly high number of live blocks i.e., L_b is above a threshold. In these cases, assigning a zero insertion age is too risky. Overall, we assign an insertion age of zero to a TC_0 block belonging to bin b with positive UC if

$$(D-L)_b > (x-1)L_b \text{ AND } L_b < \frac{3}{4} \sum_{TC=0, UC\neq 0} (L). \quad (2)$$

All the remaining TC_0 blocks with positive UC are inserted at an age of one. We will refer to this policy as the TC-UC-AGE policy. We evaluate this policy for $x = 4, 8$.

Our third insertion algorithm is similar to the TC-UC-AGE policy, but instead of assigning an age of one to all the TC_0 blocks with positive UC that do not satisfy Formula (2), it grades them from age one to three based on live population. First, the algorithm ranks the three TC=0, UC≠0 bins based on their L values and tags the bin having the smallest L value with an age of one and the one with the highest L value with an age of three. Next, the algorithm determines the bin that the incoming block belongs to and assigns the corresponding insertion age to this block. We will refer to this policy as the TC-UC-RANK policy. Unlike bypass policies, none of our insertion age assignment schemes requires dueling because a slightly wrong insertion age is not as harmful as a wrong bypass decision.

3.4 Handling Prefetches

We give some special consideration to the bins with UC=0. As we have pointed out, the blocks belonging to these bins are the result of either premature, yet correct, prefetches that failed to see a demand hit during their residency in the L2 cache or incorrect prefetches that will not see a demand hit in near future. Our bypass algorithm continues to remain oblivious to such cases and treats the UC=0 bins exactly the way it treats the other bins. Our TC-AGE insertion algorithm does not do anything special for the UC=0 blocks. The other two insertion algorithms assign a zero insertion age to a (TC=0, UC=0) block belonging to bin b if it satisfies $(D-L)_b > (x-1)L_b$ (here b is (TC=0, UC=0)). All other (TC=0, UC=0) blocks receive an insertion age of one. All ($TC_{\geq}1$, UC=0) blocks receive an insertion age of three. Figure 5 shows our bypass and TC-UC-AGE logic diagrams.

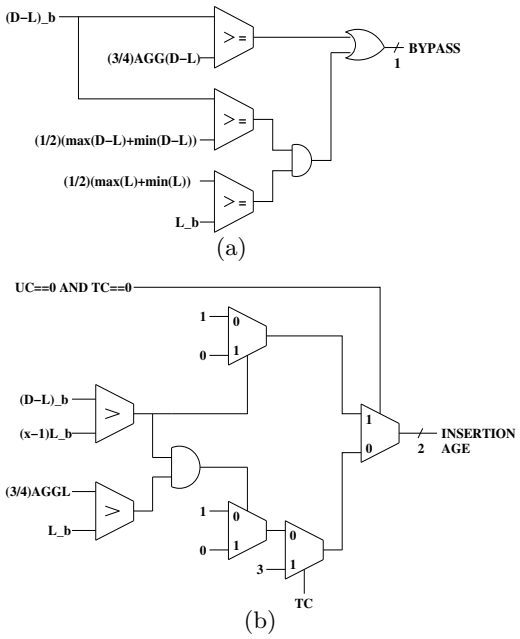


Figure 5: Logic diagrams for our (a) bypass and (b) TC-UC-AGE algorithms. $AGG(D-L)$ **refers to** $\sum_{UC\neq 0}(D-L)$ **and** $AGGL$ **refers to** $\sum_{TC=0,UC\neq 0}(L)$**. Note that the existence of an invalid way in the target set can override the bypass decision and force an insertion with age zero.**

3.5 Introducing Thread-awareness

Upgrading our bypass and insertion algorithms to a multi-threaded environment requires maintaining the $D-L$ and L counters for each TC-UC bin per thread. Each thread is also assigned a separate set of observers. The observers earmarked for a particular thread execute TC-AGE insertion for that thread and the best emerging duel winner for each of the other threads (similar to TADIP-F [16]) if bypassing is enabled. We use four observers per thread per 1024 LLC sets. Our counter update schemes do not require storage of thread id in the LLC to incorporate thread-awareness. We assume one thread per core in this article. At the time of an LLC allocation, the core id of the source L2 cache is available because this information is needed to update the coherence directory and therefore, the appropriate $D-L$ counter can be incremented. At the time of an LLC hit, the core id of the requester is available and therefore, the appropriate $D-L$ counter can be decremented and the appropriate L counter can be incremented. Also, the maximum, minimum, and aggregate values of several counters, as discussed in Section 3.1, must be maintained per thread.

4. EVALUATION METHODOLOGY

Our simulations are done on a cycle-accurate execution-driven x86 simulator. Our 4 GHz 4-way dynamically sched-

uled out-of-order issue core model closely follows the core microarchitecture of the Intel Core i7 processor [12]. Throughout this study, we assume one physical thread context per core. Each core has its own L1 and L2 caches. The L1 instruction cache is 16 KB 4-way associative and the L1 data cache is 32 KB 8-way associative. The unified L2 cache is 512 KB 8-way associative. The L2 cache is partially inclusive (also known as non-inclusive) of the L1 caches in the sense that an L2 cache eviction always queries the L1 caches for up-to-date state and data, but the L1 cache may choose to retain the block instead of invalidating. For the single-thread studies, we model a 2 MB 16-way exclusive LLC partitioned into two banks, each being 1 MB 16-way. For the multi-programming studies, we model four cores with private L1 and L2 caches and the cores are connected over a ring. Each core hop of the ring has a shared 2 MB 16-way exclusive LLC bank attached to it leading to an aggregate 8 MB 16-way shared LLC. The block size at all the three levels of the cache hierarchy is 64 bytes. We model a six-cycle hit latency (tag+data) for the L2 cache and an eight-cycle hit latency (tag+data) for each LLC bank [10]. The ring hop time is one cycle. We model a coherence directory that can accommodate eight times the number of aggregate L2 cache tags and is 16-way associative (same as the LLC). The coherence directory banks are co-located with the LLC banks. For all simulations, we model a two-channel integrated memory controller clocked at the core frequency with each channel connecting to an 8-way banked DDR3-1866 DIMM. The DRAM part (933 MHz) has burst length of 64 bits and 10-10-10 access cycle parameters. We model per-core aggressive multi-stream instruction and data prefetchers that bring blocks into the L2 cache of the core.

Our single-threaded traces are drawn from three workload categories, as already discussed: FSPEC, ISPEC, and SERVER. We first identified 213 representative dynamic code regions each of length close to thirty million dynamic instructions prefixed with a trace of several hundreds of million load/store instructions to warm up the caches. While the entire trace of about thirty million dynamic instructions is run in detailed cycle-accurate timing mode, the last six million instructions are used to measure IPC and other performance indices. All the policies evaluated in this paper are executed from the beginning of the warmup trace to make sure that the detailed cycle-accurate measurement phase captures a steady-state snapshot. Out of these 213 regions, we picked 97 regions that are likely to be sensitive to uncore optimizations (have at least five misses per kilo instructions with baseline NRF). In these 97 traces, we have 44 FSPEC traces spanning one dozen applications, namely, bwaves, cactusADM, dealII, GemsFDTD, lbm, leslie3d, milc, soplex, sphinx3, tonto, wrf, and zeusmp. We have 23 ISPEC traces spanning seven applications: bzip2, gcc, gobmk, libquantum, mcf, omnetpp, and xalancbmk. Finally, we select 30 server traces from applications like SAP, SAS, SPECjbb, SPECweb2005, TPC-C, TPC-E, etc.

We present results for 35 4-way multi-programmed workloads prepared by mixing four representative single-threaded traces from all three workload categories. Within a mix, each thread first executes its warmup region before starting the detailed performance simulation. If a thread finishes its performance simulation phase early, it continues executing so that we can model the shared LLC contention properly. The mix terminates when every thread has finished its performance simulation phase.

5. SIMULATION RESULTS

5.1 Single-threaded Workloads

We first present the simulation results with hardware data prefetchers disabled. Figure 6 summarizes the geometric mean IPC of several policies normalized to 1-bit NRF for the three single-threaded workload categories and overall (ALL). In each category, the leftmost three bars show the performance of static TC-AGE insertion and dynamic learning-based TC-UC-AGE insertion with $x = 4, 8$. To avoid unnecessarily increasing the number of policy bars, we will show the performance of TC-UC-RANK only in the presence of bypassing. The next five bars show the performance of LLC bypassing executing with three different insertion algorithms. In each of these five cases, the evaluated policy (e.g., Bypass+TC-UC-AGE-x8) is always dueled with the observers executing TC-AGE and if the observers emerge the winner, the followers disable bypassing, but continue to execute the insertion component (e.g., TC-UC-AGE-x8).

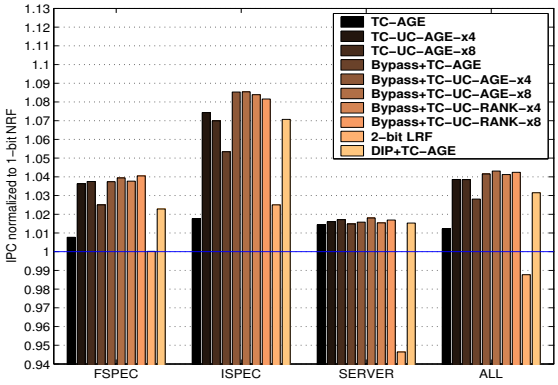


Figure 6: Summary of performance of several policies normalized to 1-bit NRF.

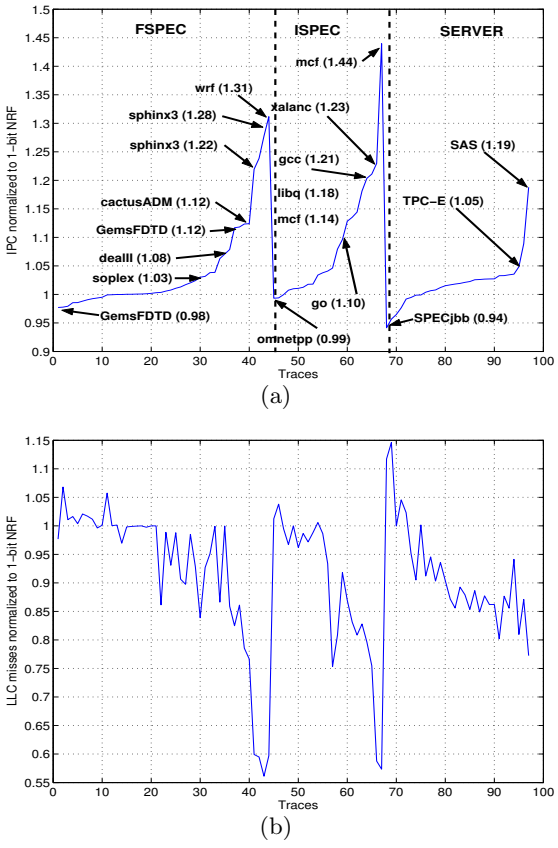
We have also experimented with a 2-bit approximation of least-recently-filled (LRF) replacement policy that ranks the blocks in a set by their fill order (can only distinguish between the last three fills). Finally, the rightmost bar in each workload category shows the performance of a dynamic insertion policy (DIP) [27] in the presence of TC-AGE insertion. This policy inserts all $TC_{\geq 1}$ blocks at age three and duels the TC_0 blocks between insertion age of zero and one. This policy shows one way to implement DRRIP-style dynamic policies originally proposed for inclusive LLCs [15].

The TC-AGE policy improves performance by more than 1% averaged over the 97 traces (see the ALL group). This result motivated us to use the TC-AGE policy for the observers in the place of NRF. The TC-UC-AGE policy improves the overall performance by almost 4%, with ISPEC showing an average performance improvement of more than 7% compared to NRF. Overall, there is no performance difference between $x = 4$ and $x = 8$ for TC-UC-AGE. Our bypass algorithm running with TC-AGE improves overall performance by 2.8%, with ISPEC showing an impressive 5.3% improvement. However, these data show that the TC-UC-AGE insertion algorithm alone can achieve better performance across the board compared to bypassing dueled with TC-AGE. Nonetheless, the Bypass+TC-AGE policy still offers an attractive design point. LLC bypassing coupled with TC-UC-AGE offers the best performance across the board with $x = 8$. The best combination i.e., Bypass+TC-UC-AGE-x8 improves the overall IPC of the 97 traces by 4.3% with FSPEC, ISPEC, and SERVER showing individual improvements of 3.9%, 8.5%, and 1.8%, respectively. Correspondingly, it saves 7.6%, 11.4%, and 8.4% of the baseline LLC misses. The IPC benefits coming from the LLC miss savings in the SERVER category are dwarfed because these workloads lose a lot of cycles in L1 instruction cache misses (even with the instruction prefetcher enabled). Overall, the Bypass+TC-UC-AGE-x8 policy requires less than 0.5% extra storage when computed as a fraction of the L2 cache and the LLC data array storage for our configuration. This overhead is summarized in Table 1.

Table 1: Summary of overhead

State	Storage (bits)	Bits
TC and UC	3 per L2 cache block	24K
LLC age	2 per LLC block	64K
Bin identity	3 per obs. LLC block (16 obs. sets per 1024 LLC sets)	1.5K
16-entry obs. CAM (per 1024 LLC sets)	10 per CAM entry (partial set index)	320
TOTAL		89.8K

The performance results for Bypass+TC-UC-RANK show that the addition of insertion age ranking mechanism based on live population does not improve beyond what Bypass+TC-UC-AGE delivers with $x = 8$. In fact, in ISPEC category, the ranking mechanism slightly hurts performance because it cannot distinguish between the TC_0 and $TC_{\geq 1}$ blocks inserted with age three. The 2-bit LRF policy improves ISPEC by 2.5%, but degrades the server workloads by 5.4%. The primary shortcoming of this policy is that a block's age in a set climbs down to zero within four fills to that set and the block becomes eligible for eviction. The 1-bit NRF policy requires a higher expected number of fills before it resets a block's age to zero (see Section 1). Finally, the DIP+TC-AGE policy improves the overall IPC by 3.2% with ISPEC improving by about 7%. Next, we analyze the performance of our best policy (Bypass+TC-UC-AGE-x8) in greater detail.

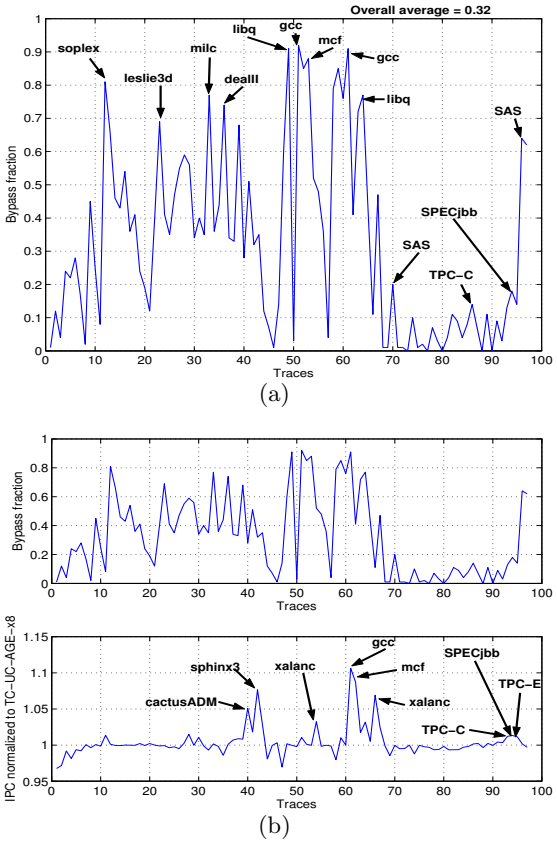


(a)

(b)

Figure 7: Distribution of (a) IPC improvements and (b) LLC misses of our best policy normalized to 1-bit NRF. We have shortened libquantum to libq.

Figures 7(a) and 7(b) respectively show the details of the IPC improvements and normalized LLC misses of individual traces running with our best LLC policy (Bypass+TC-UC-AGE-x8) compared to the baseline 1-bit NRF. The traces in each of the three categories are sorted by the IPC improvements in both the curves. Some of the traces are also marked on the curve with their IPC improvements shown within parentheses. It is important to note that different regions of the same application (e.g., GemsFDTD) react very differently to our policy, thereby emphasizing the need to simulate multiple regions of the same application. Overall, the FSPEC traces show a performance improvement of at most 31% while suffering from a performance loss of at most 2%. The ISPEC traces experience IPC improvement of up to 44% while losing at most 1% performance. The server traces show an IPC improvement of up to 19%, but also suffer from up to 6% performance losses (the poorly performing SPECjbb trace is not friendly to TC-UC-AGE). The trend in LLC misses corresponds well with the trend in IPC improvement.



(a)

(b)

Figure 8: (a) Distribution of bypass fraction in our best policy. (b) Details of additional performance gains achieved by bypassing on top of TC-UC-AGE-x8.

Next, we quantify the contributions of the LLC bypass component in our best policy (Bypass+TC-UC-AGE-x8). Figure 8(a) shows, for each trace, the fraction of L2 cache evictions bypassed by the Bypass+TC-UC-AGE-x8 policy at the time of LLC allocation. We also identify some of the application traces that show moderate to high bypass fractions. The traces are sorted exactly in the same order as in Figure 7(a). Overall, across 97 traces, on average, 32% of the L2 cache evictions are not allocated in the LLC. For FSPEC, ISPEC, and SERVER categories, the bypass percentages are 37%, 52%, and 11%, respectively.

To further quantify the performance impact of LLC bypasses in our best policy, the bottom panel of Figure 8(b) shows the IPC of Bypass+TC-UC-AGE-x8 relative to TC-UC-AGE-x8, while the top panel reproduces the bypass frac-

tion distribution for ease of comparing. Some of the application traces that enjoy noticeable benefit from LLC bypass are marked on the graph of the bottom panel. It is clear that the server traces do not enjoy much performance benefit from LLC bypasses as far as the capacity benefit is concerned. However, several FSPEC and ISPEC traces show significant improvements in IPC due to LLC bypass. A high bypass fraction does not necessarily translate to performance improvement because the retained blocks may not always have small enough reuse distances that can fit within the LLC reach. Nonetheless, our impressive bypass fraction can lead to interconnect bandwidth savings and result in further performance improvements, if our bypass scheme is implemented at the L2 cache interface.

Figure 9 shows the IPC of our best policy (Bypass+TC-UC-AGE-x8) relative to the DIP+TC-AGE policy, with several interesting trace points marked on the curve to show exactly where we gain and lose. The traces are sorted exactly the same way as in Figure 7(a). As we have already noted, we see different regions of the same application behaving differently (e.g., GemsFDTD, libquantum, SAS). Overall, while we see several traces gaining significantly compared to DIP+TC-AGE, the losses are not large.

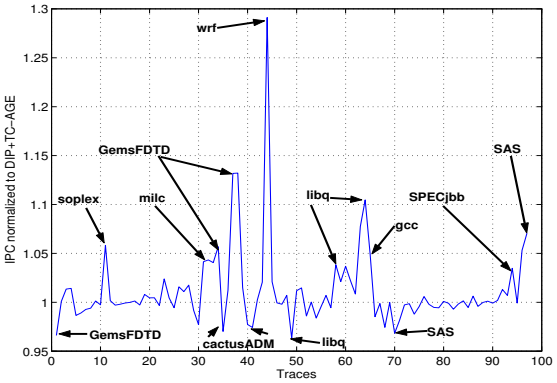


Figure 9: IPC of our best policy normalized to DIP+TC-AGE.

Figures 10(a) and 10(b) show an application-level comparison between Bypass+TC-UC-AGE-x8 and DIP+TC-AGE for SPEC 2006 and server workloads, respectively. The normalized IPC figure for each application shown in these charts is computed by taking the geometric mean of the normalized IPCs of all the traces belonging to that application. Overall, for the nineteen SPEC 2006 applications, our best policy improves IPC by 5.4% compared to 1-bit NRF, while for the eight server applications, the corresponding improvement is 1.9%. The respective improvements achieved by DIP+TC-AGE are 4.1% and 1.1%.

Finally, we turn to the performance results with an aggressive multi-stream hardware prefetcher enabled. Figure 11(a) shows the IPC improvements achieved by Bypass+TC-UC-AGE-x8 compared to the 1-bit NRF baseline with prefetchers enabled. Within each workload category, the traces are sorted by IPC improvements. Overall, for FSPEC, the IPC improvement averages at 2%; for ISPEC it is 6%; for server traces it is 4%. While the average IPC improvements for FSPEC and ISPEC have dropped compared to the non-prefetched scenario (as expected), the improvement has gone up for server traces. We find that our special handling of the UC=0 bins (see Section 3.4) helps the server traces significantly, since it is usually hard to accurately prefetch data for the server workloads. Overall, with prefetchers enabled, the IPC improvement achieved by our best policy (Bypass+TC-UC-AGE-x8) across 97 traces is 3.4%. The corresponding improvement seen by DIP+TC-AGE is 2.8%. The bypass fraction achieved by Bypass+TC-UC-AGE-x8 with the

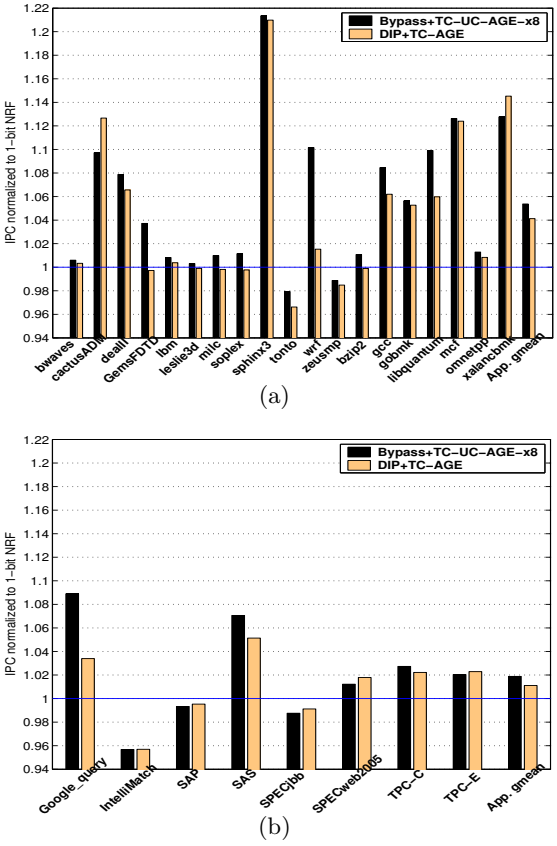


Figure 10: Details of IPC improvement achieved by our best policy and DIP+TC-AGE for (a) selected SPEC 2006 applications and (b) server applications.

prefetchers enabled is, on average, 28% of all L2 cache evictions. On the complete set of 213 traces, the average IPC improvement achieved by Bypass+TC-UC-AGE-x8 is 2.4%, with maximum slowdown being 2.8%.

Figure 11(b) further summarizes the IPC of Bypass+TC-UC-AGE-x8 normalized to that of DIP+TC-AGE in the presence of prefetching. The traces are sorted in the same way as in Figure 11(a). The traces with noticeable gains or losses are marked. Figures 12(a) and 12(b) show the application-level IPC improvements for our best policy and DIP+TC-AGE normalized to the 1-bit NRF baseline with prefetchers enabled. For the SPEC 2006 applications, our policy improves IPC by 3.7%, on average. The corresponding improvement in the server applications is 3.6%.

5.2 Multi-programmed Workloads

The results for the 4-way multi-programmed workloads are summarized in Figure 13. The left panel of Figure 13(a) evaluates the performance of three policies, namely, thread-oblivious Bypass+TC-UC-AGE-x8, thread-aware Bypass+TC-UC-AGE-x8, and thread-aware DIP+TC-AGE in terms of average IPC or throughput improvement ($\frac{\sum_i IPC_i^{\text{Policy}}}{\sum_i IPC_i^{\text{Base}}}$). The thread-aware dueling mechanism is borrowed from the TADIP-F proposal [16]. We show the performance comparison for both non-prefetched and prefetched scenarios. The right panel of Figure 13(a) quantifies the per-mix throughput improvement of thread-aware Bypass+TC-UC-AGE-x8 with prefetchers enabled. In summary, thread-awareness brings bigger performance gains in the absence of prefetching. The thread-aware Bypass+TC-UC-AGE-x8 policy improves the throughput by 2.5% in the presence of prefetch-

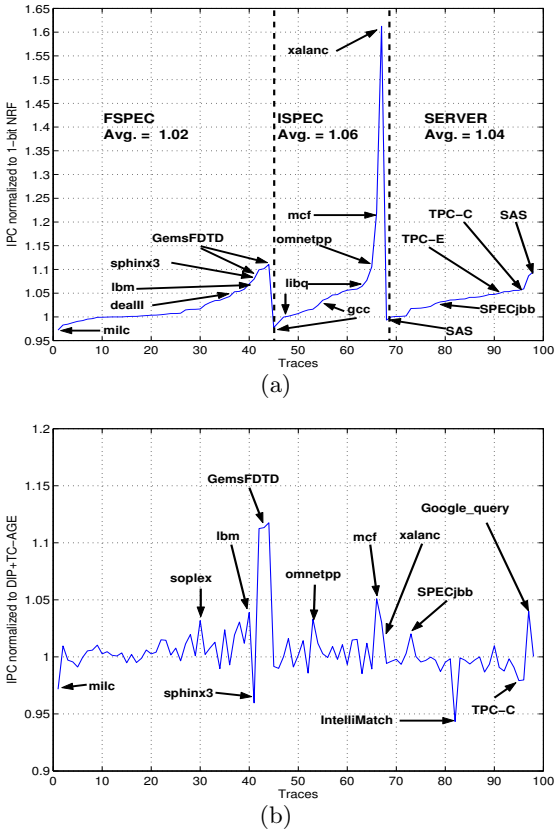


Figure 11: (a) Distribution of IPC improvements of our best policy normalized to 1-bit NRF. (b) IPC of our best policy normalized to DIP+TC-AGE. Both results are with prefetchers enabled.

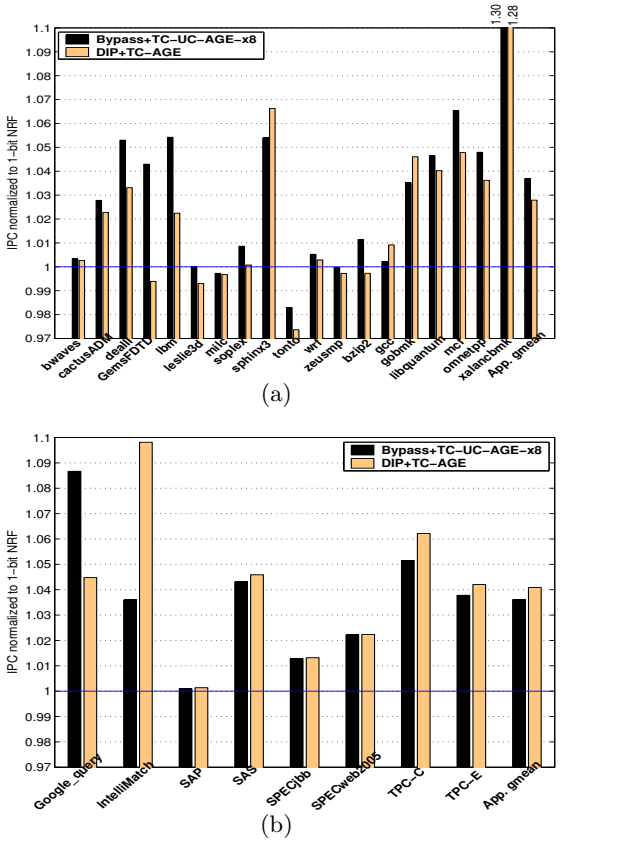


Figure 12: Details of IPC improvement achieved by our best policy and DIP+TC-AGE for (a) SPEC 2006 and (b) server applications in the presence of prefetchers.

ing, while the thread-aware DIP+TC-AGE policy improves the throughput by 1.3%.

The maximum slowdown of any individual thread should be within an acceptable range. The left panel of Figure 13(b) quantifies a conservative fairness metric $\min_i \frac{IPC_i^{\text{Policy}}}{IPC_i^{\text{Base}}}$ i.e., the normalized IPC of the slowest thread in each mix for the thread-aware Bypass+TC-UC-AGE-x8 policy with hardware prefetchers enabled. The mixes are ordered in the same way as in the right panel of Figure 13(a). Except for a few mixes, the slowdown experienced by the slowest thread is within 2% compared to the baseline and, on average, this is 1%. Finally, the right panel of Figure 13(b) details the bypass fraction achieved by thread-aware Bypass+TC-UC-AGE-x8 with hardware prefetchers enabled. While several mixes enjoy sizeable bypass fractions, the average is 9%.

6. SUMMARY

This work makes the important observation that LRU and its approximations lose meaning in exclusive LLCs and proposes a number of design choices for selective bypassing and insertion age assignment for such designs in a three-level cache hierarchy. Our LLC bypass and age assignment decisions are based on two properties of a block when it is considered for allocation in the LLC. The first one is the number of trips (trip count) made by the block between the L2 cache and the LLC from the time it is brought into the hierarchy till it is evicted from the LLC. The second property is the number of L2 cache hits (use count) experienced by a block during its residency in the L2 cache. Our best proposal is a combination of bypass and age insertion schemes based on trip count and use count, which improves the average (geometric mean) IPC of 97 single-threaded traces by 3.4% compared to a baseline not-recently-filled replacement policy in a 2 MB 16-way exclusive LLC with aggressive multi-stream prefetchers. The corresponding improvement in throughput seen by 35 4-way multi-programmed mixes is 2.5%.

7. ACKNOWLEDGMENTS

The authors thank Aravindh Anantaraman, Nithiyanandan Bashyam, Julius Mandelblat, Larisa Novakovsky, and Joseph Nuzman for useful feedback.

8. REFERENCES

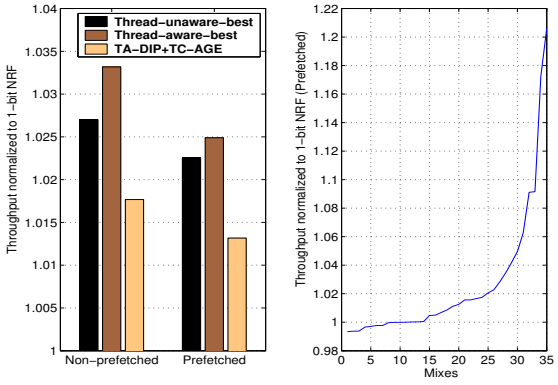
[1] Advanced Micro Devices. Family 10h AMD Opteron Processor Product Data Sheet. June 2010. support.amd.com/us/Processor_TechDocs/40036.pdf.

[2] A. Basu et al. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 421–432, December 2007.

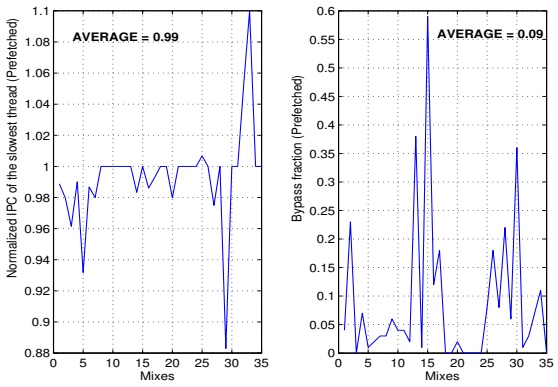
[3] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. In *IBM Systems Journal*, **5**(2): 78–101, 1966.

[4] M. Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 401–412, December 2009.

[5] J. D. Collins and D. M. Tullsen. Hardware Identification of Cache Conflict Misses. In *Proceedings*



(a)



(b)

Figure 13: (a) Throughput improvements, (b) fairness and bypass fraction for the 4-way multi-prog. workloads.

of the 32nd International Symposium on Microarchitecture, pages 126–135, November 1999.

[6] H. Gao and C. Wilkerson. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing. In *1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, June 2010.

[7] B. S. Gill. On Multi-level Exclusive Caching: Offline Optimality and Why Promotions Are Better Than Demotions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 49–65, February 2008.

[8] A. Gonzalez, C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *Proceedings of the 9th International Conference on Supercomputing*, pages 338–347, July 1995.

[9] M. Hayenga, A. Nere, and M. Lipasti. MadCache: A PC-aware Cache Insertion Policy. In *1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, June 2010.

[10] HP Labs. CACTI. Available at www.hpl.hp.com/research/cacti/.

[11] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 209–220, May 2002.

[12] Intel Corporation. Intel Core i7 Processor. www.intel.com/products/processor/corei7/index.htm.

[13] Y. Ishii, M. Inaba, and K. Hiraki. Cache Replacement Policy Using Map-based Adaptive Insertion. In *1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, June 2010.

[14] A. Jaleel et al. Achieving Non-Inclusive Cache Performance with Inclusive Caches. In *Proceedings of the 43rd International Symposium on Microarchitecture*, December 2010.

[15] A. Jaleel et al. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 60–71, June 2010.

[16] A. Jaleel et al. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques*, pages 208–219, October 2008.

[17] T. L. Johnson. Run-time Adaptive Cache Management. *PhD thesis*, University of Illinois, Urbana, May 1998.

[18] N. P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, June 1990.

[19] S. Khan, Y. Tian, and D. A. Jimenez. Sampling Dead Block Prediction for Last-level Caches. In *Proceedings of the 43rd International Symposium on Microarchitecture*, December 2010.

[20] S. Khan and D. A. Jimenez. Insertion Policy Selection Using Decision Tree Analysis. In *Proceedings of the 28th IEEE International Conference on Computer Design*, October 2010.

[21] S. Khan et al. Using Dead Blocks as a Virtual Victim Cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 489–500, September 2010.

[22] M. Kharbutli and Y. Solihin. Counter-based Cache Replacement and Bypassing Algorithms. In *IEEE Trans. on Computers*, **57**(4): 433–447, April 2008.

[23] A-C. Lai, C. Fide, and B. Falsafi. Dead-block Prediction & Dead-block Correlating Prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, June/July 2001.

[24] K. M. Lepak and R. D. Isaac. Mostly Exclusive Shared Cache Management Policies. *US Patent 7640399*, Advanced Micro Devices, Inc. (Sunnyvale, CA, US), December 2009. www.freepatentsonline.com/7640399.html.

[25] H. Liu et al. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 222–233, November 2008.

[26] R. L. Mattson et al. Evaluation Techniques for Storage Hierarchies. In *IBM Systems Journal*, **9**(2): 78–117, 1970.

[27] M. K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[28] G. Tyson et al. A Modified Approach to Data Cache Management. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 93–103, November/December 1995.

[29] Y. Zheng, B. T. Davis, and M. Jordan. Performance Evaluation of Exclusive Cache Hierarchies. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–96, March 2004.